# Declarations and definitions

# Declarations and definitions

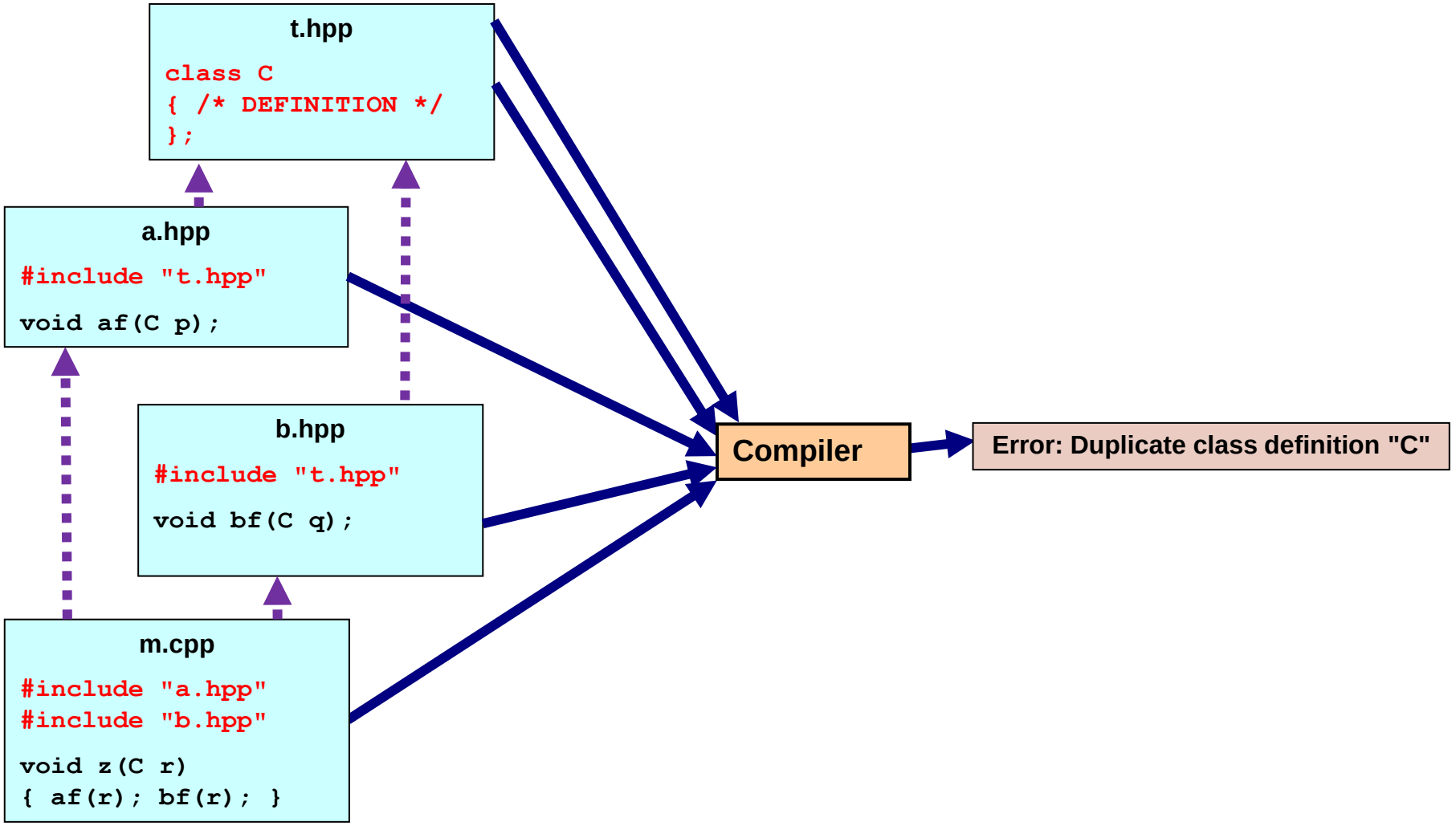
- Declaration
  - A construct to declare the existence (of a class/variable/function/...)
    - Identifier
    - Some basic properties
    - Ensures that (some) references to the identifier may be compiled
      - Some references may require definition
- Definition
  - A construct to completely define (a class/variable/function/...)
    - Class contents, variable initialization, function implementation
    - Ensures that the compiler may generate runtime representation
  - Every definition is a declaration
- Declarations allow (limited) use of identifiers without definition
    - Independent compilation of modules
    - Solving cyclic dependences
    - Minimizing the amount of code that requires (re-)compilation

# Declarations and definitions

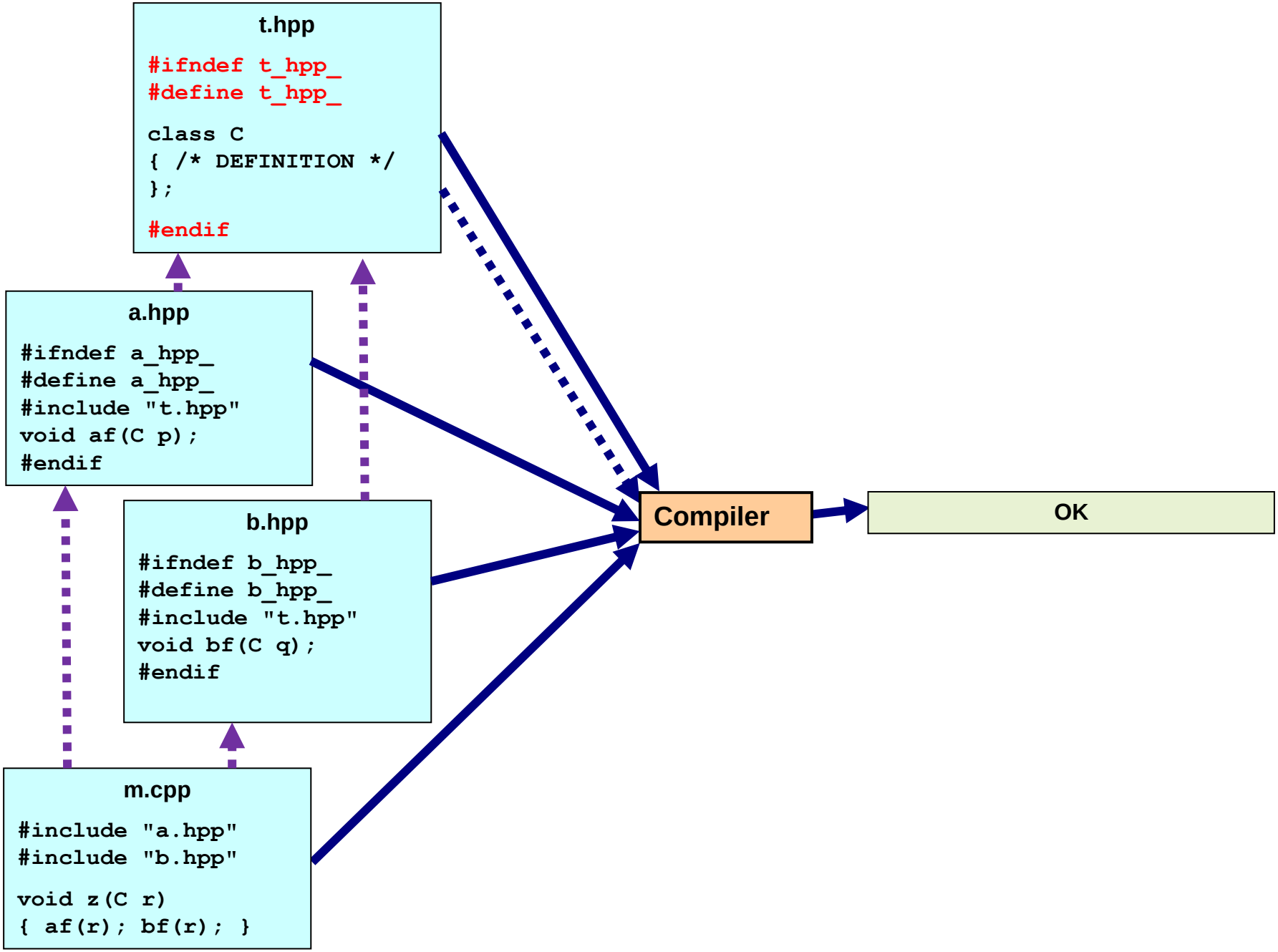
- One-definition rule #1:
  - One *translation unit*...
    - (*module*, i.e. one .cpp file and the .hpp files included from it)
  - ... may contain at most one definition of any item

- One-definition rule #2:
  - Program...
    - (i.e. the .exe file including the linked .dll files)
  - ... may contain at most one definition of a non-inline variable or function
    - Definitions of classes, types, inline variables or inline functions may be contained more than once (due to inclusion of the same .hpp file in different modules)
      - If these definitions are not identical, undefined behavior will occur
      - Beware of version mismatch between headers and libraries
    - Diagnostics is usually poor (by linker)

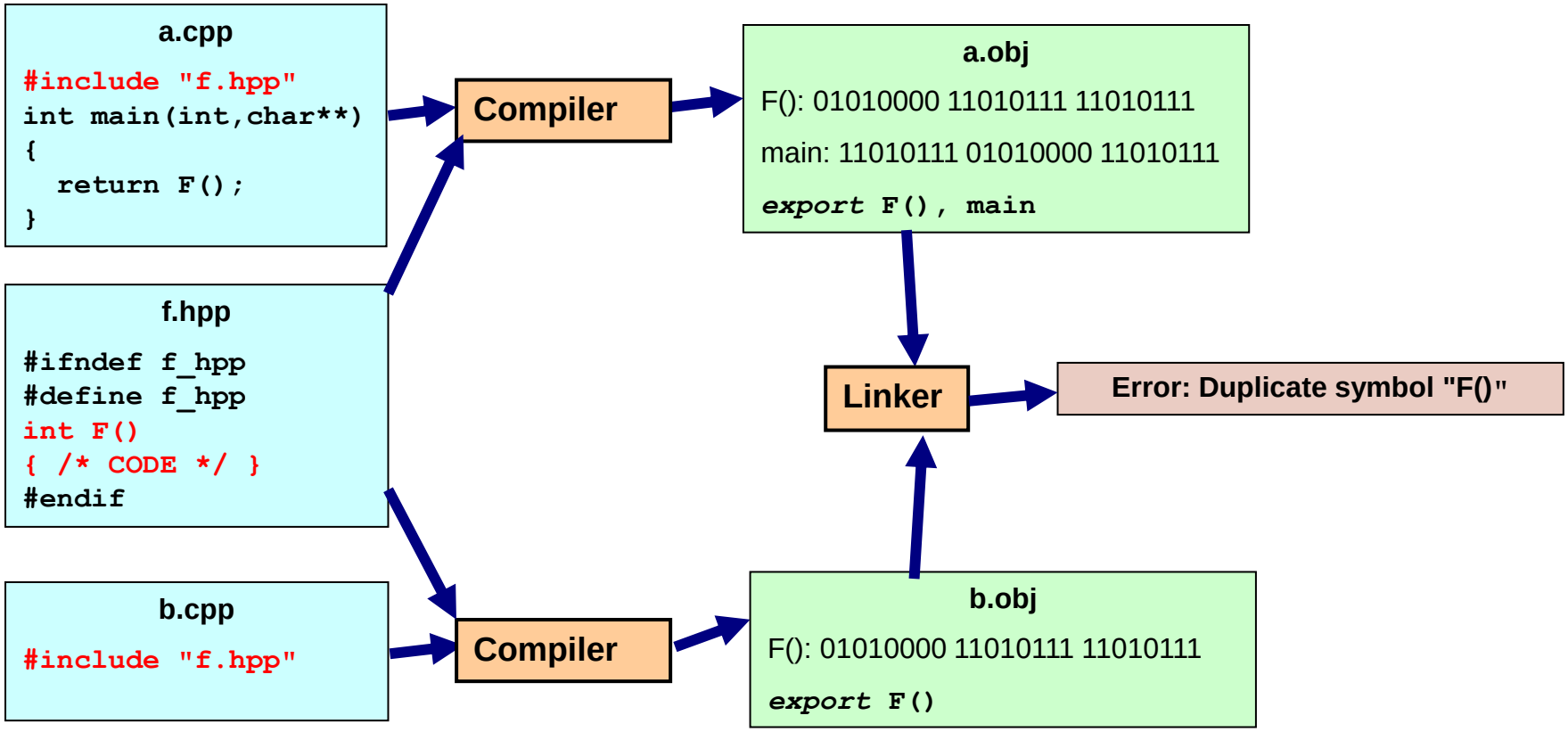
# ODR #1 violation



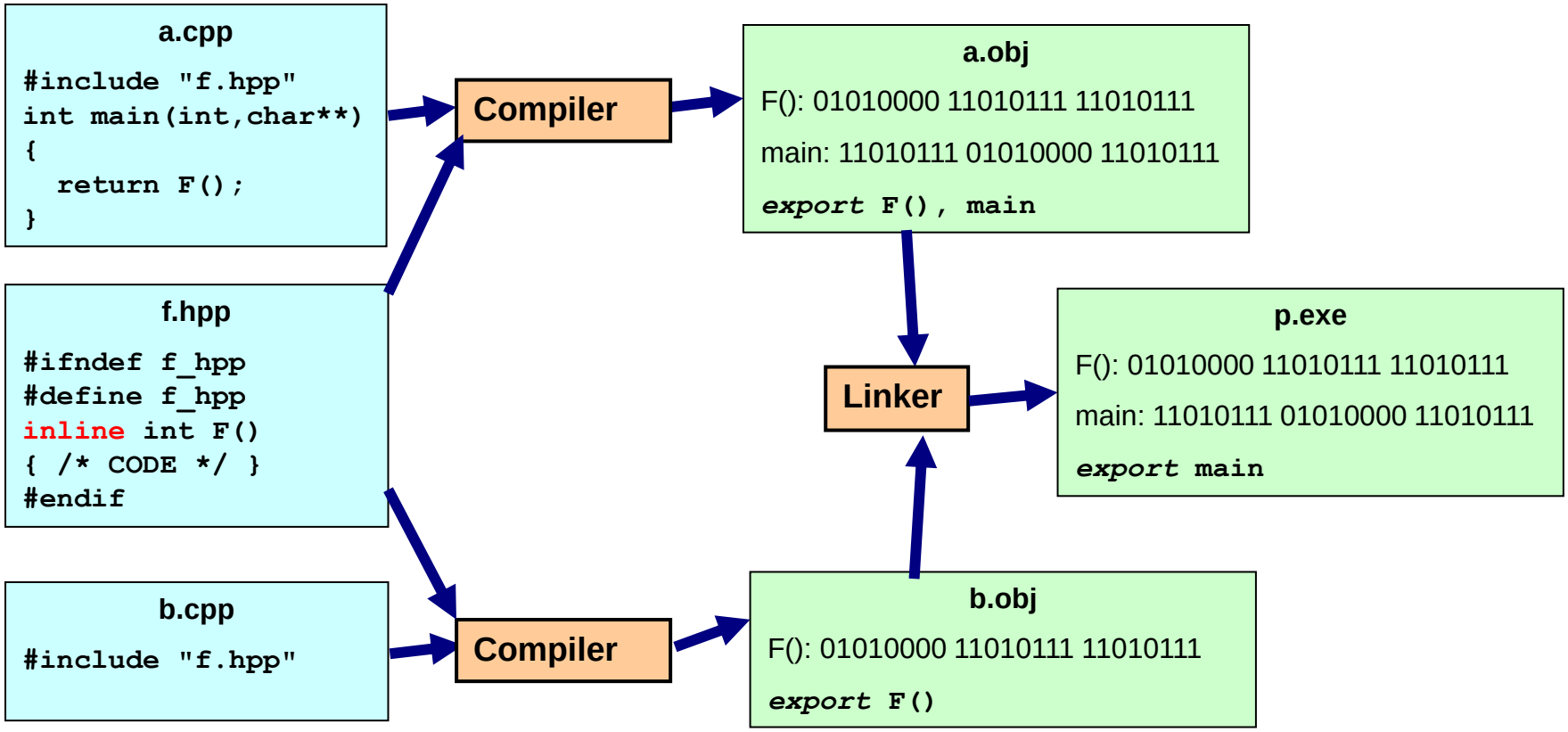
# ODR #1 protection



# ODR #2 violation



# ODR #2 protection



# Placement of declarations

- Every name must be declared **before** its first use
  - In every *translation unit* which uses it
  - "Before" refers to the text produced by inclusion and conditional compilation directives
- Special handling of member function bodies
  - Compilation of the body of a member function...
    - if the body is present inside its class definition
  - ... is delayed to the end of its class definition
    - thus, declarations of all class members are visible to the body
- The placement of declaration defines the scope of the name
  - A declaration always uses an unqualified name
  - The definition of a previously declared item may use qualified name
    - Member function definitions outside its class
    - Namespace member definitions outside its namespace
- Exception: Friend functions
  - Friend function declaration inside a class declares the name outside the class (if not already declared)
  - Beware: There are some consequences wrt. Argument-Dependent Lookup

# Placement of declarations

```
class C {
public:
   D f1();          // error: D not declared yet
   int f2() { D x; return x.f3(); }    // OK, compilation delayed
   class D {
   public:
      int f3();
   };
   friend C f4();  // declares global f4 and makes it a friend
private:
   int m_;
};
C::D C::f1() { return D{}; }          // qualified name C::D required outside C
int C::D::f3() { return 0; }          // this could be static member function
void C::f5() {}    // error: cannot declare outside the required scope
C f4() { C x; x.m_ = 1; return x; }  // friends may access private members
```

# Placement of definitions

- Type alias (typedef/using), enumeration type, constant
  - Must be defined **before** first use (as seen after preprocessing)
- Class/struct
  - Class/struct C must be defined **before** its first **non-trivial** use:
    - (member) variable definition of type C, inheriting from class C
    - creation/copying/moving/destruction of an object of type C
    - access to any member of C
  - **Trivial** use is satisfied with a declaration
    - constructing complex types (C*,C&,C&&,C(),T(C),C[]) from C
    - *declaring* functions accepting/returning C
    - manipulating with pointers/references to C
- Inline function, inline global/static-member variable [C++17]
  - must be defined anywhere in **each** translation unit which contains a call
    - the definition is typically placed in a .hpp file
- Non-inline function, non-inline global/static-member variable
  - must be defined exactly **once** in the program (if used)
    - the definition is placed in a .cpp file
- Static function, static global variable
  - independent existence in each translation unit which contains a declaration
    - the declaration/definition is placed in a .cpp file
    - considered obsolete, use anonymous namespaces if needed
    - placement in a .hpp file is usually a nonsense – use inline instead

# Cyclic references in code

## Class A refers to B

```
struct A {
  A( int p) : v{p} {}
  B generate_b()
  {
    return B(this);
  }
  int v;
}
```

- Declaration of generate_b requires declaration of B
  - Therefore definition of A requires declaration of B
- Definition of generate_b requires definition of B

## Class B refers to A

```
struct B {
  B( A * q) : link{q} {}
  int get_v()
  {
    return link->v;
  }
  A * link;
}
```

- Declarations of constructor and link require declaration of A
  - Therefore definition of B requires declaration of A
- Definition of get_v requires definition of A

# Cyclic references in code

```
struct B;
struct A {
  A( int p) : v{p} {}
  B generate_b();
  int v;
}
struct B {
  B( A * q) : link{q} {}
  int get_v()
  {
    return link->v;
  }
  A * link;
}
inline B A::generate_b()
{
  return B(this);
}
```

- A correct ordering
  - There are more possible
- Declaration of B
- Definition of A
  - Except definition of generate_b
- Definition of B
- Definition of generate_b

# Cyclic references between headers

- A.hpp

```
#ifndef A_hpp_
#define A_hpp_
#include “B.hpp”
struct A {
  A( int p) : v{p} {}
  B generate_b();
  int v;
}
inline B A::generate_b() {
  return B(this);
}
#endif
```

- WRONG!
  - When A.hpp is compiled, ifndef guards prohibit recursive A.hpp inclusion from B.hpp
    - Definition of B will not see the definition of A

- B.hpp

```
#ifndef B_hpp_
#define B_hpp_
#include “A.hpp”
struct B {
  B( A * q) : link{q} {}
  int get_v() {
    return link->v;
  }
  A * link;
}
#endif
```

- WRONG!
  - When B.hpp is compiled, ifndef guards prohibit recursive B.hpp inclusion from A.hpp
    - Definition of A will not see the declaration of B

# Cyclic references between headers

- A.hpp

```
#ifndef A_hpp_
#define A_hpp_
struct B;
struct A {
  A( int p) : v{p} {}
  B generate_b();
  int v;
}
#include “B.hpp”
inline B A::generate_b() {
  return B(this);
}
#endif
```

- A ticket to a madhouse
  - Never bury include directives inside header or source files

- B.hpp

```
#ifndef B_hpp_
#define B_hpp_
#include “A.hpp”
struct B {
  B( A * q) : link{q} {}
  int get_v() {
    return link->v;
  }
  A * link;
}
#endif
```

- STILL WRONG!
  - B.hpp includes TOO MUCH
    - A.hpp contains the definition of generate_b which cannot compile before the definition of B

# Cyclic references between headers

- A.hpp

```
#ifndef A_defined
#define A_defined
struct B;
struct A {
  A( int p) : v{p} {}
  B generate_b();
  int v;
}
#endif
#ifndef generate_b_defined
#define generate_b_defined
#include “B.hpp”
inline B A::generate_b() {
  return B(this);
}
#endif
```

- B.hpp

```
#include “A.hpp”
#ifndef B_hpp_
#define B_hpp_
struct B {
  B( A * q) : link{q} {}
  int get_v() {
    return link->v;
  }
  A * link;
}
#endif
```

- It works, but your colleagues will want to kill you
  - Never use define guards for anything else than **complete** header files
    - including the include directives

# Cyclic references between classes – solved with .cpp files

- A.hpp

```
#ifndef A_hpp_
#define A_hpp_
struct B;
struct A {
  A( int p) : v{p} {}
  B generate_b();
  int v;
}
#endif
```

- A.cpp

```
#include “A.hpp”
#include “B.hpp”
B A::generate_b() {
  return B(this);
}
```

- B.hpp

```
#ifndef B_hpp_
#define B_hpp_
#include “A.hpp”
struct B {
  B( A * q) : link{q} {}
  int get_v() {
    return link->v;
  }
  A * link;
}
#endif
```

- Still problematic
  - Including A.hpp enable you to call generate_b which returns undefined class B

# Cyclic references between classes – solved with more .hpp files

- Atypes.hpp

```
#ifndef Atypes_hpp_
#define Atypes_hpp_
struct B;
struct A {
  A( int p) : v{p} {}
  B generate_b();
  int v;
}
#endif
```

- A.hpp

```
#ifndef A_hpp_
#define A_hpp_
#include “Atypes.hpp”
#include “B.hpp”
inline B A::generate_b() {
  return B(this);
}
#endif
```

  - Never include Atypes.hpp or Btypes.hpp directly
    - Except in the corresponding A.hpp and B.hpp

- Btypes.hpp

```
#ifndef Btypes_hpp_
#define Btypes_hpp_
struct A;
struct B {
  B( A * q) : link{q} {}
  int get_v() {
    return link->v;
  }
  A * link;
}
#endif
```

- B.hpp

```
#ifndef B_hpp_
#define B_hpp_
#include “Btypes.hpp”
#include “A.hpp”
inline int B::get_v() {
  return link->v;
}
#endif
```

  - Instruct everybody to include A.hpp or B.hpp
    - Otherwise they may miss some inline definition
      - Which results in linker error if called

# Cyclic references between templates – solved with more .hpp files

- Atypes.hpp

```
#ifndef Atypes_hpp_
#define Atypes_hpp_
template< typename T> struct B;
template< typename T> struct A {
  A( T p) : v{p} {}
  B<T> generate_b();
  T v;
}
#endif
```

- A.hpp

```
#ifndef A_hpp_
#define A_hpp_
#include “Atypes.hpp”
#include “B.hpp”
template< typename T> inline B<T> A<T>::generate_b() {
  return B<T>(this);
}
#endif
```

  - Never include Atypes.hpp or Btypes.hpp directly
    - Except in the corresponding A.hpp and B.hpp

- Btypes.hpp

```
#ifndef Btypes_hpp_
#define Btypes_hpp_
template< typename T> struct A;
template< typename T> struct B {
  B( A<T> * q) : link{q} {}
  T get_v() {
    return link->v;
  }
  A<T> * link;
}
#endif
```

- B.hpp

```
#ifndef B_hpp_
#define B_hpp_
#include “Btypes.hpp”
#include “A.hpp”
template< typename T> inline T B<T>::get_v() {
  return link->v;
}
#endif
```

  - Instruct everybody to include A.hpp or B.hpp
    - Otherwise they may miss some inline definition
      - Which results in linker error if called